Chapter 3 Overview of Research

This dissertation proposes, specifies, investigates, and evaluates an approach to provide automated assistance to designers of concurrent software for real-time applications. The current chapter gives an overview of this research. Section 3.1 presents an evaluation of existing approaches to automate the generation of software designs. Then, section 3.2 discusses, from the designer's perspective, the process of generating concurrent designs for software systems. Section 3.3 identifies specific objectives for the research reported in this dissertation. Section 3.4 proposes and describes a knowledge-based approach to automate the generation of concurrent designs for real-time software. Section 3.5 outlines a means to evaluate the proposed approach.

3.1 Evaluation of Existing Approaches

As discussed in the previous chapter, research into approaches for designing software produced a number of design methods, commonly used today among software designers. One of the most well-known of these methods begins by expressing the system under design as a behavioral model based upon flow graphs that show how data enters the system, is transformed, and then leaves the system. Such flow graphs are known as data flow diagrams, or DFDs. After producing a satisfactory DFD model of the software system, the designer then allocates elements of the DFD to a set of software

modules and defines the relationships among those modules. The designer then expresses the resulting software modules and calling relationships using a structure chart.

A number of researchers have addressed the problem of generating a software architecture from a behavioral model of a software system. These research efforts, described previously in Chapter 2 and presented in summary form in Table 1, exhibit a number of limitations that leave room for additional research. Most of the existing research addresses the generation of structure charts from DFDs, but does not address software systems modeled with data/control flow diagrams (D/CFDs) and control transformations, as permitted, for example, by the Real-Time Structured Analysis (RTSA) method.

In addition, all but one of the research efforts summarized in Table 1 produce a sequential design. The SARA research effort generates concurrent designs by examining a separate specification, in the form of system verification diagrams, or SVDs, that indicate the parallelism existing among the elements on a related DFD. No research considers generating a concurrent design from a single flow graph. Another limiting aspect of the approach taken to generate designs for SARA is that no design method underlies the approach. Rather, the decision method for the SARA design generator uses rules that are defined uniquely for mapping SVD elements to objects in the Graph Behavior Model (GMB) that underlies the SARA design simulator and for mapping the DFD directly to the SARA structural model. This decision method limits the approach to the SARA design environment.

Table 1. Summary of Current Research on Automating Software Design Methods

	Existing Research Effort			
Trait	CAPO	EARA	STES	SARA
Input Model	DFDs	DFDs	DFDs	DFDs & SVDs
Output Model	Structure Charts	Structure Charts	Structure Charts	GMB Objects & SARA Structural Model
Decision Method	Coupling & Cohesion	Structured Design	Structured Design	Mapping Rules
Underlying Techniques	Clustering Algorithms	Formal Rule Rewriting	Expert System Rules	Expert System Rules
Completeness & Consistency Checking	No	No	No	No
Traceability	Implicit	Explicit	Implicit	Implicit
Design Rationale Capture	No	No	No	No
Requires Elicitation From Designer	No	Yes	Yes	Yes
Varies Elicitation With Designer's Experience	No	No	No	No
Varies Design With Target Environment	No	No	No	No
Connects With CASE Tool	No	No	Yes	Yes

Another limitation of the current research derives from the underlying techniques used to automate the decision-making. Among the approaches in Table 1 two underlying techniques appear: clustering algorithms and rules. Software designers use methods based on a range of knowledge, not all of which can be conveniently and naturally represented by rules or by clustering algorithms. For example, semantic relationships among elements in a DFD cannot be represented and reasoned about easily with rules or with clustering algorithms. Completeness and consistency checking, which cannot be implemented with clustering algorithms, can be implemented with rules; however, more natural techniques, such as assertion monitoring or database query systems, are available. None of the research identified in Table 1 provides a means for automatically checking a generated design for completeness and consistency.

In another limitation, the existing approaches lack support for explicit traceability between the input specification and the resulting design. Only one of the approaches, EARA, provides explicit traceability from the DFD to the structure chart. The other approaches provide implicit traceability by allowing a designer to manually examine the design for the presence of elements from the input specification. Such implicit traceability proves unsatisfactory for all but the simplest of designs. In a related shortcoming, the existing approaches fail to capture the rationale for design elements. Given a decision-making method that automatically generates a design, the possibility exists to capture both the design decisions and the reasons for those decisions. The capture of design rationale, essential to the success of an automated design generator, can

help a human designer to understand the generated design. Allowing the designer to analyze rule traces, as suggested by Lor and Berry for the SARA approach, appears insufficient.

Three of the four existing approaches, summarized in Table 1, elicit information, in several forms, from a human designer during design generation. For example, some approaches ask a designer for additional information, when required to make design decisions. Some approaches present a designer with design choices that depend upon a designer's preference or judgment. Some approaches also present a designer with tentative design decisions and then ask the designer to confirm or override those decisions. Despite this wide range of consultations that might be taken with a designer, none of the existing approaches varies the elicitation based upon a designer's level of experience.

The current approaches exhibit one final limitation worth noting. None of the approaches can vary the generated designs to account for variations in target environment. Significant characteristics in a target environment might include, for example, the availability of shared memory, the availability of message queuing services, the number of processors, and the number of signals permitted between tasks. Many of the significant characteristics in a target environment become important when constructing a concurrent design but remain of marginal importance when producing a sequential design. Since they always produce sequential designs, the CAPO, EARA, and STES approaches do not consider characteristics of the target environment. The CAPO

approach does, however, generate alternate designs by using a variety of clustering algorithms. The design generator for SARA, intending always to produce designs executed on the SARA design simulator, also fails to consider variations among target environments.

3.2 Generating Concurrent Designs for Real-Time Software

A more effective approach to automate the generation of software designs might be found by considering the design process from a software designer's perspective. The software designer aims to transform a set of informal, software requirements into a complete and precise structure for a set of software components and relationships that can satisfy the stated requirements and that can guide the remainder of the software development process through detailed design, coding, testing, and documentation. This job is critical to the success of a software project. Before an architectural design exists, project costs include only the funds needed to gather and document the requirements. After an architectural design exists, project costs increase rapidly, as the staff grows to include designers, coders, testers, and technical writers. In addition, mistakes made, but not identified and corrected, during the initial design of a software system usually have far-reaching consequences, requiring extensive reworking of software, tests and documentation.

3.2.1 The Designer's Objectives

To successfully produce an architectural design, sometimes referred to as a high-level design, a designer must achieve six objectives.

- First, the designer must analyze and improve, where necessary, the software requirements. The designer must ensure that the requirements, both functional and nonfunctional, are complete, consistent, precise, and correct. For functional requirements, the designer analyzes the problem and typically produces some sort of behavioral model from the informal requirements. Often, a designer uses an approach based on a combination of techniques, such as flow diagrams, state-transition diagrams, data modeling methods, and scenario analysis. For nonfunctional requirements, such as performance and reliability targets, the designer creates a measurable specification for each requirement.
- Second, the designer must define the software structure. To meet this objective, the designer must identify all necessary software components and the relationships among them. For all designs, this includes software modules and their calling relationships and parameter passing details. For concurrent designs, the designer must also identify tasks and establish the interface mechanisms among the tasks. When both tasks and modules exist in the same design, the designer must also define the relationships among the tasks and modules. Generally, the designer depicts the software structure using a graphical representation. For sequential designs, structure charts provide a useful representation. For concurrent designs, an architectural diagram, using one of several available notations, can provide a concise, visual representation of the software structure.

- Third, the designer must specify the software components and relationships present in the design. The designer uses two forms of specification. One form documents the relevant details of each component. For modules this might include: the module name, any information hidden within the module, the basis for producing the module, any assumptions about the module, a list of anticipated changes, and a specification of the operations provided and used by the module. For each operation provided by a module, the specification usually includes a list of parameters. For tasks in a concurrent design, a specification might include: the task name, the events, messages, and data accepted and generated by the task, and modules invoked by the task, the basis for forming the task, any timing constraints and estimates of execution time, a list of errors detected by the task, and an outline of the task's behavior. The second form of specification, often overlooked when producing a software design, details the design history for each component.
- Fourth, the software designer must ensure that the evolving design considers and properly accounts for details of the intended target environment. For example, the hosting operating system might not provide message queuing services. In such a case, whenever a design requires queued messages, the designer must introduce into the design an alternative method for those messages among tasks.
- Fifth, the software designer must evaluate the design. Evaluation can take many forms. Certainly, the designer must evaluate the design against both the functional and nonfunctional requirements to ensure they can be satisfied. In addition,

especially for concurrent designs, the designer must check the design against a set of implicit requirements, such as, freedom from deadlock and livelock, fairness among tasks that share resources, and boundedness of behavior. Another form of evaluation might consider the relative merits and deficiencies among a set of alternate designs.

- Sixth, the designer must configure the design. This can take various forms. In one form, the designer might be required to allocate software components among multiple processors in a multiprocessor system or among network nodes in a distributed system. Given a set of tasks assigned to a specific processor, the designer must also establish execution priorities among the tasks. In a second form, the designer might map the software components to specific software packaging constructs provided by the target environment.

To consistently meet these six objectives, the designer might choose from among a number of software design methods. Some software design methods lead to sequential designs, while others lead to concurrent designs. This dissertation proposes a knowledge-based approach to automate the generation of concurrent designs; therefore, a software design method for concurrent and real-time systems provides the knowledge underlying the approach.

3.2.2 A Software Design Method for Concurrent and Real-Time Systems

Gomaa defines a software design method for concurrent and real-time systems. [Gomaa93] Gomaa's method consists of two main components. The first component,

Concurrent Object-Based Real-time Analysis, or COBRA, enables the designer to construct a behavioral model of the functional requirements for a software system. The second component, COconcurrent Design Approach for Real-Time Systems, or CODARTS, provides heuristics that help a designer transform a COBRA behavioral model into a concurrent design, consisting of tasks and modules and the relationships among them. Together, COBRA and CODARTS define knowledge that enables a designer to produce concurrent designs of reasonable quality on a repeatable basis. The highlights of COBRA and CODARTS are discussed, in turn, below.

3.2.2.1 COBRA

COBRA provides the designer with an approach to model the functional requirements of a real-time application. COBRA represents behavioral models with the notation from Real-Time Structured Analysis, as defined by Ward and Mellor, [Ward85] but with some restrictions and extensions, noted, where applicable, in the following discussion of each component of a COBRA model.

The COBRA environmental model depicts, using a system context diagram, the boundary between the software system and its external environment. In addition, COBRA provides criteria for decomposing a large system into separate subsystems. Each subsystem can then be represented with a subsystem context diagram. COBRA allocates the terminators from the system context diagram among the subsystem context diagrams, as appropriate. In addition, each subsystem context diagram includes an additional terminator for each other subsystem with which the subject subsystem

interacts. COBRA uses terminators to represent user roles. This triple usage of terminators to represent devices, subsystems, and user roles represents one of COBRA's extensions to RTSA.

COBRA further decomposes each context diagram, either system or subsystem, as appropriate, into a hierarchical set of data/control flow diagrams. At the leaves of this hierarchy, COBRA represents control objects, encapsulating state-transition diagrams, with RTSA control transformations, while representing both non-control objects and functions with RTSA data transformations. This use of data transformations to represent both objects and functions represents another of COBRA's extensions to RTSA. COBRA uses pseudo-code to specify the detailed behavior associated with non-control objects and functions. COBRA represents data repositories using RTSA data stores, along with any associated data dictionaries.

COBRA uses RTSA data flows and control flows to depict data and events, respectively, flowing among the elements of a data/control flow diagram. COBRA defines syntactic conventions to establish a semantic significance for certain event flows. For example, event flows labeled as trigger, enable, and disable represent control of a data transformation by a state-transition diagram within a control transformation. In addition, COBRA models timer events using event flows that have no apparent source. These conventions represent additional extensions and restrictions to RTSA.

COBRA also introduces the idea that certain transformations, representing objects, on a data/control flow diagram might be classified by type. For example, Gomaa

identifies device input/output objects, user role objects, control objects, data abstraction objects, and algorithm objects. Given the type of an object, a designer might make some semantic inferences about the object. In COBRA, Gomaa also proposes to classify functions by type and to identify some relationships between objects and functions in a data/control flow diagram. For example, Gomaa distinguishes among functions provided by objects, functions managed by control objects, and functions, both asynchronous and periodic, operating independently.

In addition to these conventions for modeling a problem with RTSA notation, COBRA also introduces a technique called behavioral analysis to help a designer construct an appropriate set of data/control flow diagrams and related state-transition diagrams. Behavioral analysis provides the designer with a set of steps that lead to creation of an appropriate COBRA model for a real-time problem.

3.2.2.2 CODARTS

To transform a COBRA specification into a concurrent design, the designer must consider the transformations in the specification from two different perspectives. From one perspective the designer looks for transformations that might execute independently as tasks. Typically this involves identifying transformations that interact with asynchronous and periodic devices, often called edge transformations, followed by considering which of the remaining, internal, transformations should form the basis for separate threads of control. After identifying a candidate set of transformations that might become independent tasks the designer then considers whether some of the

transformations in the candidate set might be combined into the same task. Once a suitable set of tasks is identified the designer then considers how the remaining transformations, that is, those that did not form the basis for a task, can be allocated among the tasks. From a second perspective the designer looks to combine transformations and data stores from the specification into information-hiding modules. Here the designer might consider which transformations provide operations on data stores, which provide interfaces to devices, which hide state-transition diagrams, and which hide algorithms. After determining both the tasks and information-hiding modules (IHMs) in a concurrent design, the designer then decides which IHMs are shared by multiple tasks and which are accessed solely by one task. These decisions about module sharing lead to a preliminary architecture for the design. Subsequently, the designer defines the interfaces among tasks, among tasks and IHMs, and among IHMs.

Gomaa provides a set of heuristics that help a designer to transform a COBRA specification into a concurrent design. These heuristics, collectively called CODARTS, address four main objectives: structuring tasks, defining interfaces among tasks, structuring modules, and integrating the task and module views of the design.

3.2.2.2.1 Task Structuring

Task structuring in CODARTS requires a designer to examine the data/control flow diagrams, and any related supporting specifications such as state-transition diagrams, pseudo-code, and data dictionaries, produced during COBRA modeling, along with any additional textual specification that might be pertinent, in an effort to group

transformations into tasks in a concurrent design. To help with this process, CODARTS provides a set of task-structuring criteria.

Several subsets of the task-structuring criteria enable a designer to establish concurrent tasks based on selected transformations from a data/control flow diagram. In general, two types of tasks can be identified: device input/output tasks and internal tasks. Using CODARTS task-structuring criteria for device input/output tasks, a designer creates tasks based on periodic devices, based on asynchronous devices, and based on resource monitoring needs. Using CODARTS task-structuring criteria for internal tasks, a designer allocates tasks based on needs for periodic processing and for processing of asynchronous requests. In addition, the designer might identify some tasks based on the existence of multiple instances of the same transformation. For example, one instance of a task that controls an elevator might be defined for each elevator that exists.

Another subset of the task-structuring criteria, known as task-cohesion criteria, enables a designer to combine transformations from a data/control flow diagram into the same concurrent task. CODARTS provides a number of cohesion criteria. Temporal cohesion suggests that a designer combine transformations into the same task when the transformations must execute with identical frequency or when they can execute with frequencies that share a common factor. Sequential cohesion suggests that a designer combine a set of transformations into the same task when the members of that set cannot execute concurrently, either because a strict sequencing requirement exists or because the execution among the transformations is mutually exclusive. Control cohesion suggests

that a designer combine data transformations triggered by a control transformation into the same task as the control transformation, provided that the processing associated with the data transformations completes during a state transition within the control transformation. Functional cohesion suggests that a designer combine transformations in the same task when the processing associated with the transformations performs logically related functions. Another form of cohesion, task inversion, suggests that, under certain circumstances, a designer invert a multiple-instance task into a single-instance task with context-switching performed within the inverted task. The application of these cohesion criteria can require significant judgment on the part of the designer, especially because the criteria must be considered together and because the criteria can conflict.

3.2.2.2.2 Task-Interface Definition

A number of mechanisms exists for communicating between pairs of tasks and for communicating between tasks and the external environment. The most prominent mechanisms include: input and output data, timer events, external events, software signals, and loosely-coupled and tightly-coupled messages, both with and without reply. In addition, loosely-coupled messages might be assigned various priorities to enable a receiving task to distinguish the importance of messages within a message queue. A designer must map elements from the data/control flow diagram to task interfaces, while at the same time selecting specific mechanisms to implement each interface. Often, the selection of specific mechanism depends upon the services available in the intended

target environment. CODARTS provides guidelines for performing the required mapping of elements and selection of mechanisms.

3.2.2.2.3 Module Structuring

Similar to task structuring, module structuring in CODARTS requires a designer to examine the data/control flow diagrams in a effort to group transformations and data stores into information hiding modules. To help with this process, CODARTS provides a number of module-structuring criteria based on information hiding.

One criterion forms a device-interface module for each device interface object on the data/control flow diagram. Another criterion forms a data-abstraction module for each data store on the data/control flow diagram, where multiple transformations access the data store. Once a data-abstraction module exists, a designer uses a number of criteria to allocate functions to the module based upon the relationship between each function and a data store within the module. A third criterion maps each control object on the data/control flow diagram to a state-transition module. Another criterion forms a function-driver module for each set of data transformations that emit data flows to the same device input/output object. A fifth criterion forms an algorithm-hiding module from each transformation that represents an application-specific algorithm.

CODARTS also provides some guidelines for determining module operations for modules of specific types, including: device-interface modules, data-abstraction modules, state-transition modules, and function-driver modules. In some cases

CODARTS gives explicit guidance, while in other cases a designer can derive guidance from considering a number of examples given by Gomaa.

3.2.2.2.4 Integrating Tasks and Modules

Given both the task and module views of a concurrent design, the designer must consider how the tasks and modules can be integrated to form a completed software architecture. Integration requires the designer to distinguish between cases where only one task accesses a module and cases where several tasks share access to a module. Graphically, the designer places each module accessed by a single task into the accessing task, while placing shared modules outside any task. The reader should view these module placement decisions as logical decisions, not as decisions about the physical packaging of modules and tasks. The designer considers physical packaging later, when configuring the design. CODARTS provides guidelines for making decisions about the logical placement of modules.

After placing modules, relative to tasks, in the design, the designer must identify each case where a task invokes an operation in a module external to the task. In addition, the designer must identify each case where an operation within a module outside any task requires an operation within another module outside any task. These dependencies identify the calling paths among tasks and operations in external modules. CODARTS establishes that these dependencies must be identified, but provides little explicit guidance as to a method.

3.2.2.2.5 Configuration and Evaluation

Gomaa provides additional guidance for configuring and evaluating concurrent designs. CODARTS includes some rules-of-thumb for categorizing tasks based upon time-criticality and then for assigning relative priorities based upon this categorization. Gomaa also specifies strategies for mapping a concurrent design to an Ada environment and for mapping designs to distributed and multiprocessor systems. Gomaa describes three methods of evaluating designs against performance objectives. Two methods, event-sequence analysis and real-time scheduling theory, enable a designer to perform a static analysis of the performance characteristics of a concurrent design. The third method requires the designer to construct a simulation model of the design and then to conduct simulation experiments to perform a dynamic analysis of the design.

3.3 The Research Problem

As an overall goal, the research reported in this dissertation aims *to develop an effective method for automating the generation of concurrent designs for real-time software, given a flow graph model of system behavior*. In addition, the proposed method attempts to satisfy the following objectives:

- use heuristics, from an existing design method, for designing both tasks and information hiding modules,
- provide two-way traceability between the data/control flow diagram and the resulting design,

- provide a basis for checking the completeness, with respect to the input specification, and the self-consistency of the resulting design,
- capture the rationale for each design decision,
- allow alternative designs to be generated from the same specification based upon variations in the intended target environment,
- elicit information from a designer only when such information is essential and cannot be inferred, and
- vary decision-making responsibility depending upon the designer's level of experience.

To accomplish the overall goal and to satisfy the listed objectives, this dissertation provides solutions for the following, specific, research problems:

1. Modeling and Analyzing Specifications,
2. Modeling Designs and Target Environments, and
3. Modeling Decision-Making Processes.

Each of these research problems is explained in the following sections.

3.3.1 Modeling and Analyzing Specifications

The RTSA notation provides a small set of symbols for constructing a data/control flow diagram. Yet, when creating and contemplating such a diagram, a human designer views various combinations of symbols with semantic significance. This semantic view of the diagram exists only in the designer's head. Later, when generating a concurrent design from the flow diagram, the designer uses this semantic view to

consider decisions about task and module structuring, about task-interface definition, and about task and module integration. In addition, the designer understands which of the many semantic concepts implicit in the diagram require additional information to improve design decisions. In most such cases, the designer also supplies the missing information at the appropriate time.

To help a human designer, then, an automated reasoning tool must somehow share the same semantic understanding of a data/control flow diagram as the designer. Where possible, an automated tool should infer the semantic meaning of symbols on a data/control flow diagram. In addition, where needed or helpful to generate a design, an automated tool must understand what information to elicit from a designer .

The research problem, then, is to define and specify a model for representing and reasoning about specifications written using RTSA notation. This model must permit RTSA data/control flow diagrams to be provided as input specifications, must enable semantic content to be inferred from data/control flow diagrams, and must permit representation of any additional information required to make design decisions. Inference of semantic content from a data/control flow diagram should be automated to the greatest extent possible. Chapter 4, augmented in Appendix A with a complete specification of the axioms and classification rules used in the model, gives the proposed solution for this problem.

3.3.2 Modeling Designs and Target Environments

While many design decisions involve only the mapping of semantic concepts on a data/control flow diagram to elements in a concurrent design, some significant decisions, as well as consistency and completeness checking, require reasoning about the evolving, concurrent design itself. In addition, certain design decisions entail knowledge about specific characteristics of the intended target environment. For these reasons, an automated designer's assistant needs some understanding of concurrent designs and target environments. *The research problem is twofold: 1) to define and specify a design model and 2) to define a model for describing target environments.* The design model must enable representation of, and reasoning about, concurrent designs, including both tasks and modules and their relationships. The design model must provide for traceability to and from a data/control flow diagram, must allow design decisions to be associated with each element in the model, must facilitate completeness checking of generated designs against input data/control flow diagrams, and must allow generated designs to be checked for consistency against the design model. A second model must permit a designer to represent significant characteristics of target environments. Chapter 5 describes the proposed solution for each of these problems.

3.3.3 Modeling Decision-Making Processes

Generating designs involves a process of making decisions and then assessing the results of those decisions. In some situations the order of decision-making has significance, while in other situations the order is irrelevant. Sometimes, a number of

possible decisions can conflict. As explained above when discussing the CODARTS method, the designer applies a semantic understanding of a data/control flow diagram, and related specifications, knowledge of the evolving design, and an understanding of salient traits of the target environment to construct a concurrent design, decision-by-decision: structuring tasks, defining task interfaces, structuring modules, and integrating tasks and modules.

To assist a designer in the decision-making required to generate a concurrent design for real-time software, a number of problems must be solved. *One problem is to define and specify the design-decision rules that compose each design phase.* The research reported in this dissertation uses four design phases, as identified in CODARTS: task structuring, task-interface definition, module structuring, and task and module integration. *A related, but subsidiary, problem requires that the design-decision rules, composing each of the four design phases, be grouped into decision-making processes, based upon the purpose of each rule, and that any ordering restrictions required among the various rules within each group be identified and specified. In addition, any ordering required among the decision-making processes within each design phase must be specified.* To satisfy an objective of the research reported in this dissertation, the design-decision rules must reflect heuristics, from an existing design method, for designing both tasks and modules. To meet this objective, the proposed solution uses heuristics from the CODARTS design method. In addition, design-decision rules that represent subtle or difficult decisions requiring consultation with a human designer must

only be referred to an experienced designer. When an experienced designer is unavailable, difficult decisions should be taken without consulting the designer. Chapter 6 contains the proposed solution for task structuring. Chapters 7, 8, and 9 propose solutions for task-interface definition, module structuring, and task and module integration, respectively.

3.4 A Knowledge-based Approach to Design Generation

This dissertation proposes a knowledge-based approach to solve the problems described in sections 3.3.1, 3.3.2, and 3.3.3. An overview of the proposed approach follows.

3.4.1 A Conceptual View of the Approach

Figure 1 provides a conceptual view of a knowledge-based approach to assist a designer to generate a concurrent design for real-time software. The conceptual view identifies and depicts relationships among three meta-models, two sets of meta-knowledge, and four knowledge bases. Together, the three meta-models define the semantic concepts and semantic relationships that can be reasoned about when generating a design. In essence, the knowledge contained within the three meta-models enables the modeling of models of: 1) real-time problems, 2) concurrent designs, and 3) target environments; thus, the choice of the term meta-model. Figure 1 depicts, using directed arcs, consultations made by the various knowledge bases to the meta-models. The Specification Meta-Model, specified in Chapter 4, can be used to reason about data/control flow diagrams, represented using syntactic elements from Real-Time

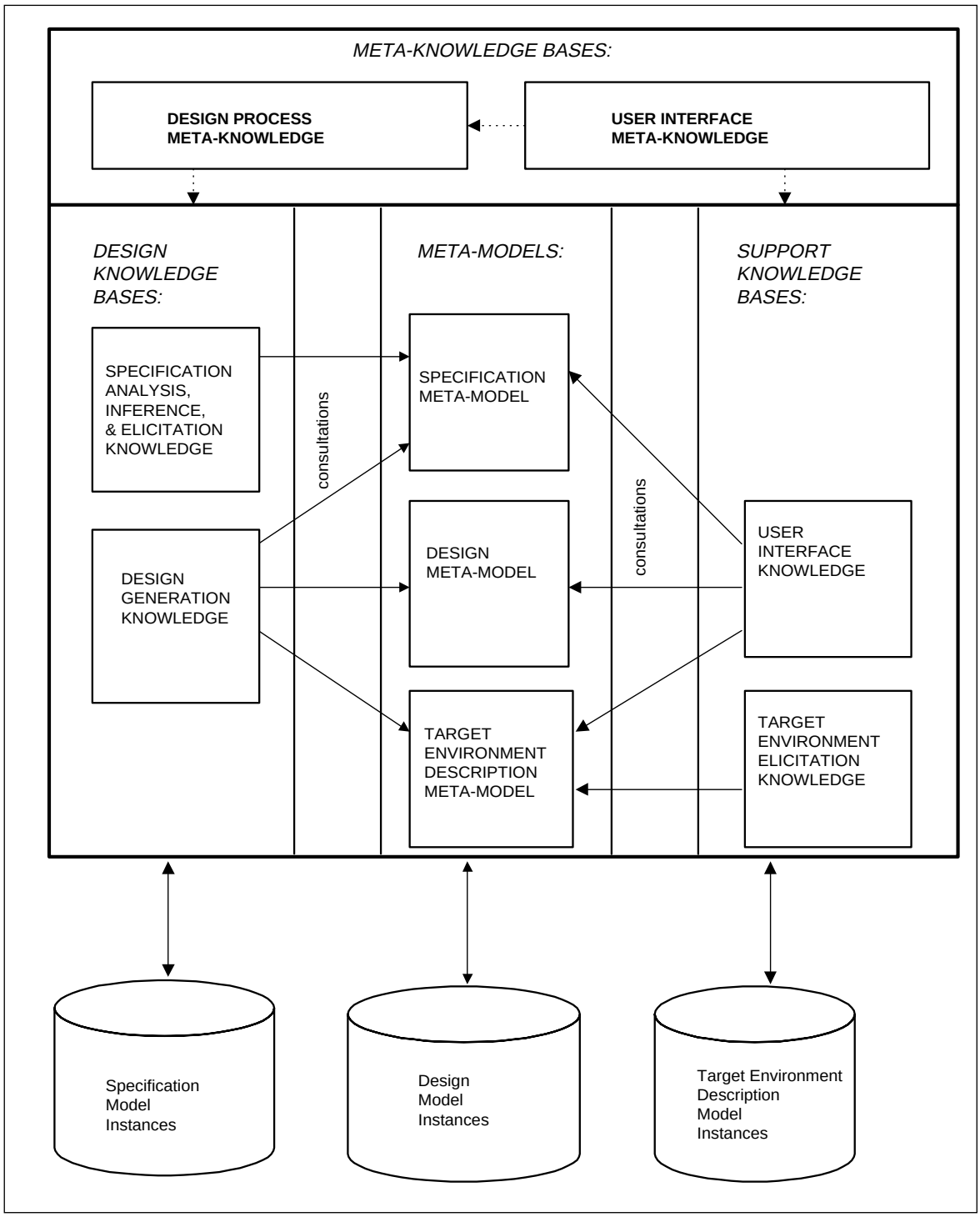

Figure 1. Proposed Conceptual Architecture for Knowledge-based Generation of Concurrent Designs for Real-Time Software

Structured Analysis (RTSA), and to represent certain specification addenda that cannot be represented with RTSA. The Specification Meta-Model builds upon ideas for restricting and extending the use of RTSA notation, as described by Gomaa in his COBRA method. [Gomaa93, Chapter 13] The Design Meta-Model and the Target Environment Description Meta-Model, specified in Chapter 5, provide a means of representing and reasoning about concurrent designs and target environments, respectively. In addition, the Design Meta-Model provides a foundation for checking the completeness and consistency of generated designs.

Two knowledge bases, illustrated in Figure 1, contain expertise needed to solve specific facets of the design problem. Chapter 4 contains a specification and discussion of the expertise composing the Specification Analysis, Inference, and Elicitation Knowledge. This expertise can be decomposed into four knowledge bases, enumerated below.

- Concept Classification Knowledge defines how to classify the syntactic elements of an RTSA data/control flow diagram as semantic concepts within the Specification Meta-Model.
- Classification Checking Knowledge defines the means for checking whether or not each syntactic element in a RTSA data/control flow diagram is classified fully as a leaf-level, semantic concept within the Specification Meta-Model.

- Axiom Checking Knowledge defines the method for checking whether or not each semantic concept in a given data/control flow diagram satisfies appropriate axioms, specified as a part of the Specification Meta-Model.
- Information Elicitation Knowledge defines the means to recognize when additional information is required from a designer and includes the mechanisms for eliciting the needed information.

A second knowledge base shown in Figure 1, Design Generation Knowledge, contains the expertise required to generate a concurrent design, consisting of tasks and modules and their relationships and interfaces. This expertise can be decomposed into six knowledge bases, one for each phase of the CODARTS method.

- Task Structuring Knowledge, as specified in Chapter 6, defines how to allocate transformations from a data/control flow diagram to tasks in a concurrent design.
- Task Interface Definition Knowledge, as specified in Chapter 7, defines how to construct interfaces among the tasks in a concurrent design.
- Module Structuring Knowledge, as specified in Chapter 8, defines how to allocate transformations and data stores from a data/control flow diagram to software modules.
- Task and Module Integration Knowledge, as specified in Chapter 9, defines how to relate tasks and modules in a concurrent design.
- Design Configuration Knowledge, which remains outside the scope of this dissertation, provides a placeholder for a method to help a designer configure a

concurrent design for execution on particular target environments, such as multiprocessor systems with and without shared memory, or on distributed systems.

- Design Evaluation Knowledge, which remains outside the scope of this dissertation, provides another placeholder for a method to help a designer statically and dynamically assess the performance and correctness characteristics of a concurrent design.

Two additional knowledge bases, shown in Figure 1, provide some ancillary expertise, not strictly related to the design-generation process. These knowledge bases are not discussed in this dissertation, except as they apply to the description of a proof-of-concept prototype, recounted in Chapter 10. The Target Environment Elicitation Knowledge defines the means to elicit and record target environment descriptions provided by a designer. The User Interface Knowledge contains expertise to perform various operations that help a designer to manage an automated design environment. This includes knowledge about how to:

- list known specifications and target environments,
- load, save, and discard specifications and target environment descriptions,
- checkpoint and restore evolving designs, and
- query specifications, target environment descriptions, and designs.

Two meta-knowledge bases, shown in Figure 1, orchestrate the use of the various knowledge bases. Meta-knowledge is knowledge about knowledge. Whereas knowledge

defines **how** to solve specific problems, meta-knowledge defines **when** to solve specific problems and also **who** can solve specific problems. The User Interface Meta-Knowledge defines the control mechanisms for interacting with the designer, and for activating the Design Process Meta-Knowledge. The User Interface Meta-Knowledge, incidental to the research described in this dissertation, is discussed in this chapter only as it relates to the initiation of the design-generation process. Any other discussion of the User Interface Meta-Knowledge is deferred until Chapter 10, where a proof-of-concept prototype is described. The Design Process Meta-Knowledge ensures that components of the Design Knowledge Bases are applied in an effective order so that concurrent designs can be generated. The Design Process Meta-Knowledge is presented in further detail below in order to provide a more comprehensive understanding of the relationships between the various Design Knowledge Bases.

3.4.1.1 Design-Process Meta-Knowledge

Figure 2 depicts the relationships among the various meta-knowledge bases and selected components of the design and support knowledge bases from Figure 1. The User Interface Meta-Knowledge base is shown, along with only two components from the User Interface Knowledge base. Specification Loading Knowledge and Target Environment Description Loading Knowledge establish a specification instance and a target environment description, respectively, as the current focus for consideration. The Design Process Meta-Knowledge from Figure 1 is divided in Figure 2 into two partitions.

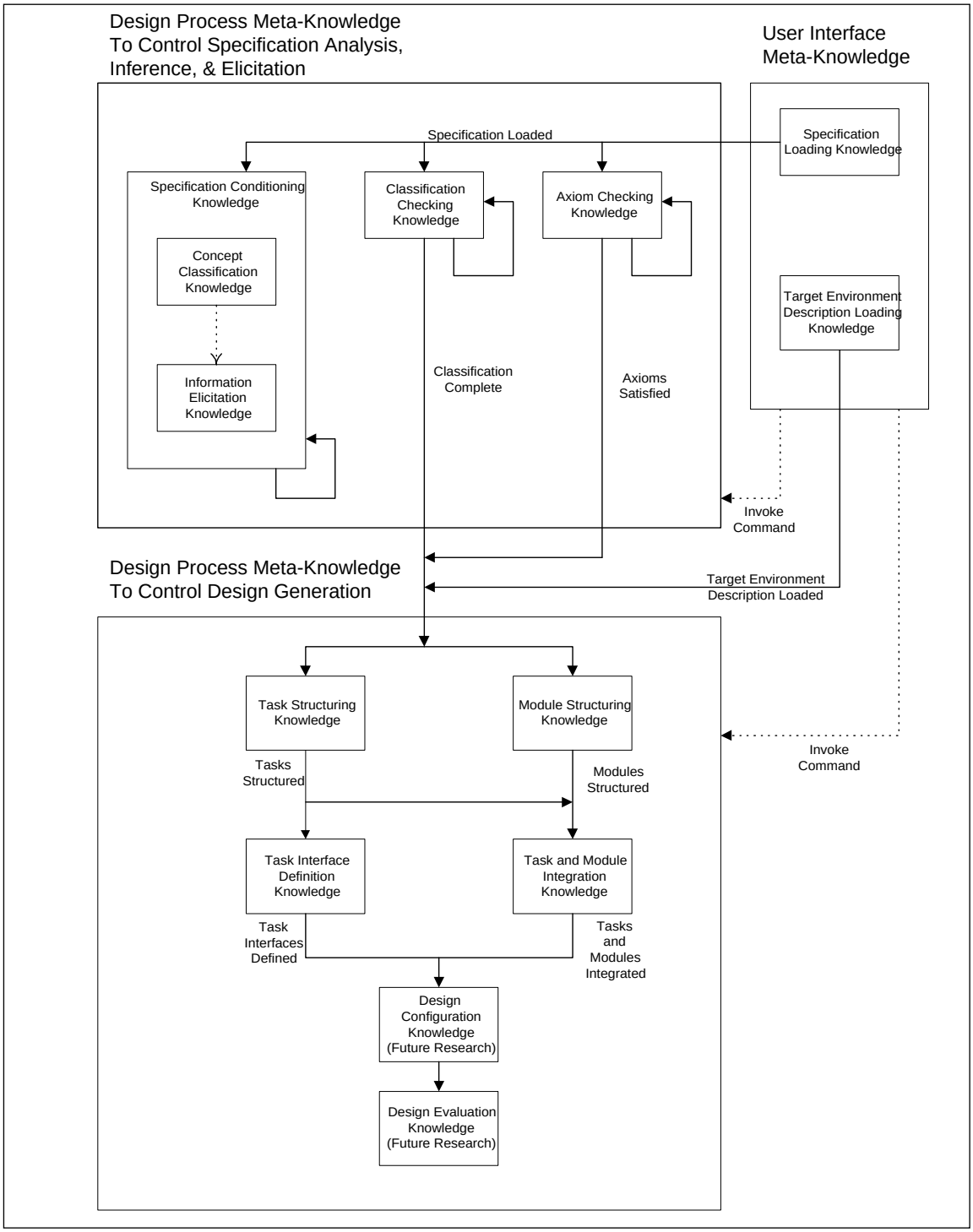

Figure 2. Decomposition of the Design-Process Meta-Knowledge and Related Knowledge Bases

The first partition of the Design Process Meta-Knowledge controls the application of three components of the Specification Analysis, Inference, and Elicitation Knowledge base: Specification Conditioning Knowledge, Classification Checking Knowledge, and Axiom Checking Knowledge. Any of these components can be invoked, repeatedly and at any time, through the User Interface Meta-Knowledge, provided that a sole precondition is satisfied: a specification must have been loaded. The component, Specification Conditioning Knowledge, is further decomposed into two subordinate knowledge bases: Concept Classification Knowledge and Information Elicitation Knowledge. Whenever the Specification Conditioning Knowledge is invoked, these two subordinate knowledge bases are executed in the order shown. This ordering constraint exists because the classification of certain concepts, such as Timers and Interrupts, creates a requirement to elicit additional information, such as period and maximum arrival rate, from the designer. The first partition of the Design Process Meta-Knowledge, then, consists of three simple constraints.

1. A specification must be loaded before any component of the Specification Analysis, Inference, and Elicitation Knowledge base is invoked.
2. A component of the Specification Analysis, Inference, and Elicitation Knowledge base must be invoked through the User Interface Meta-Knowledge.
3. Whenever Specification Conditioning Knowledge is invoked, then the two subordinate knowledge bases, Concept Classification Knowledge and Information Elicitation Knowledge, must be executed, and in that order.

Where feasible, meta-knowledge is expressed, as described above, through constraints, represented as preconditions. This approach avoids the imposition of sequential ordering on decision-making when no such ordering is necessary. For example, the first partition of the Design Process Meta-Knowledge could be defined as a strict ordering among the relevant knowledge components:

- Specification Loading Knowledge,
- Concept Classification Knowledge,
- Information Elicitation Knowledge,
- Classification Checking Knowledge, and then
- Axiom Checking Knowledge.

However, save for the need to first load a specification and the need to invoke the two Specification Conditioning Knowledge components, Concept Classification and Information Elicitation Knowledge, as an integral, sequential unit, the ordering would be overly constrained.

Assuming the preconditions are satisfied, the invocation of each knowledge component in the Specification Analysis, Inference, and Elicitation Knowledge base yields some result. For Axiom Checking, the result will be either: Satisfied or Unsatisfied. For Classification Checking, the result will be either: Complete or Incomplete. For Specification Conditioning, the result will be a specification that might be updated based on automated inferences or on information elicited from an analyst.

Some of these results satisfy preconditions in the second partition of the Design Process Meta-Knowledge.

The second partition of the Design Process Meta-Knowledge controls the application of components of the Design Generation Knowledge base. The Design Generation Knowledge base consists of six, decision-making knowledge bases, as shown in Figure 2, each corresponding to a phase in the design-generation process. The detailed content and specific organization of four of these knowledge bases make up the core aspects of this dissertation, as described in Chapters 6 through 9. The remaining two knowledge bases, which are outside the scope of this dissertation, are discussed in Chapter 12 as future research. These six, decision-making knowledge bases, governed by specified preconditions, generate a concurrent design from a valid specification.

The Task Structuring Knowledge, which is based on the CODARTS task structuring criteria described by Gomaa [Gomaa93, Chapter 14], determines tasks within the evolving design. The Module Structuring Knowledge uses the CODARTS module structuring criteria [Gomaa93, Chapter 15] to formulate information hiding modules. The Task and Module Integration Knowledge decides which modules belong inside tasks and which must be placed outside of tasks; this knowledge also determines which modules are invoked by tasks and which modules are clients of other modules. To place modules, relative to tasks, heuristics for integrating tasks and information hiding modules are drawn from the CODARTS design method [Gomaa93, Chapter 16]. The Task Interface Definition Knowledge draws on CODARTS heuristics for identifying external task

interfaces and for determining task communication and synchronization [Gomaa93, Chapter 14]. These heuristics help to establish the type of interfaces required between tasks within the design and to specify the messages received and sent by tasks. The Task Interface Definition Knowledge also uses CODARTS heuristics for task packaging [Gomaa93, Chapter 17] as a basis to allocate inter-task queues to appropriate mechanisms.

The Design Configuration Knowledge retains the responsibility for assigning tasks to processors and for assigning priorities to tasks. The Design Evaluation Knowledge tests the resulting design against a set of assertions that the designer intended to satisfy. This knowledge also allows the design to be analyzed, statically and dynamically, for acceptable performance. The specification of Design Configuration and Evaluation Knowledge is outside the scope of the research reported in this dissertation.

As shown in Figure 2, these six, decision-making, knowledge bases must meet certain ordering constraints among themselves. More specifically, the Task Structuring and Module Structuring Knowledge can be invoked independently, but both tasks and modules must be structured before the Task and Module Integration Knowledge can be initiated. The Task Interface Definition Knowledge can be invoked at any time after tasks are structured. These preconditions, taken together, impose minimal ordering constraints on the design-generation process.

In addition to the internal ordering constraints, the six, decision-making, knowledge bases must meet a few other constraints. First, each decision-making knowledge base must be invoked through the User Interface Meta-Knowledge. Second,

before either the Task or Module Structuring Knowledge base can be invoked, a specification must be loaded, be classified fully, and have all axioms satisfied, and a Target Environment Description must be loaded.

Table 2 distills the foregoing discussion into a few concepts. The Design Process Meta-Knowledge controls nine, decision-making knowledge bases used to analyze specifications and to generate designs. Each decision-making knowledge base can be invoked individually through the User Interface Meta-Knowledge, which then refers to the Design Process Meta-Knowledge to ensure that all necessary preconditions are satisfied before allowing the decision-making knowledge base to be used. Expressing Design Process Meta-Knowledge as preconditions embedded within the invocation mechanism for each decision-making knowledge base increases flexibility because the design-generation process is not encumbered with artificial ordering constraints. Since each decision-making knowledge base will not be invoked until its preconditions are satisfied, a designer can attempt to invoke design steps in any order that seems convenient. Should the preconditions for a particular step prove unsatisfied, the designer can be so informed and can then take some appropriate action.

This flexibility might not, however, prove advantageous to the novice designer who might prefer a simple, fixed order that is known to be safe, that works, and that does not require a detailed understanding of the design-generation process. To account for such needs, the decision-making knowledge bases can be packaged in a specific order, which meets the preconditions imposed by the Design Process Meta-Knowledge, and

Table 2. Design-Process Meta-Knowledge Expressed as Preconditions on Decision-Making Knowledge Bases

Decision-Making Knowledge Base	**Preconditions**
Specification Conditioning	1. Specification Loaded
Classification Checking	1. Specification Loaded
Axiom Checking	1. Specification Loaded
Task Structuring	1. Specification Loaded 2. Axioms Satisfied 3. Classification Complete 4. Target Environment Description Loaded
Module Structuring	1. Specification Loaded 2. Axioms Satisfied 3. Classification Complete 4. Target Environment Description Loaded
Task and Module Integration	1. Tasks Structured 2. Modules Structured
Task Interface Definition	1. Tasks Structured
Design Configuration	1. Task Interfaces Defined 2. Tasks and Modules Integrated
Design Evaluation	1. Design Configured

then invoked with one command. For example, in the prototype described in Chapter 10, a command, **generate_design**, invokes, in order, the following knowledge bases: Specification Conditioning Knowledge, Classification Checking Knowledge, and Axiom Checking Knowledge, Task Structuring Knowledge, Task Interface Definition Knowledge, Module Structuring Knowledge, and Task and Module Integration Knowledge. Using this approach, the novice designer views design generation as a one-step process, where the automated assistant prompts the designer as information is required. The more experienced designer can view the design-generation process in finer detail, including the abilities to review progress along the way, to discard unsatisfactory results, to revise the specification, and to save intermediate results that can be restarted later. A precondition-based approach to expressing Design Process Meta-Knowledge allows both of these views to coexist within the same automated design generator.

3.4.1.2 Decision-Making Knowledge Bases and Design-Decision Rules

To comprehend the proposed method for generating concurrent designs, further consideration must be given to the internal composition of each decision-making knowledge base that composes the Design Generation Knowledge. Each of these decision-making knowledge bases is assigned a specific design goal. To achieve its assigned goal, each decision-making knowledge base is divided into a sequence of decision-making processes, where each decision-making process is assigned a subgoal in support of the overall goal for the knowledge base. Figure 3 illustrates this general

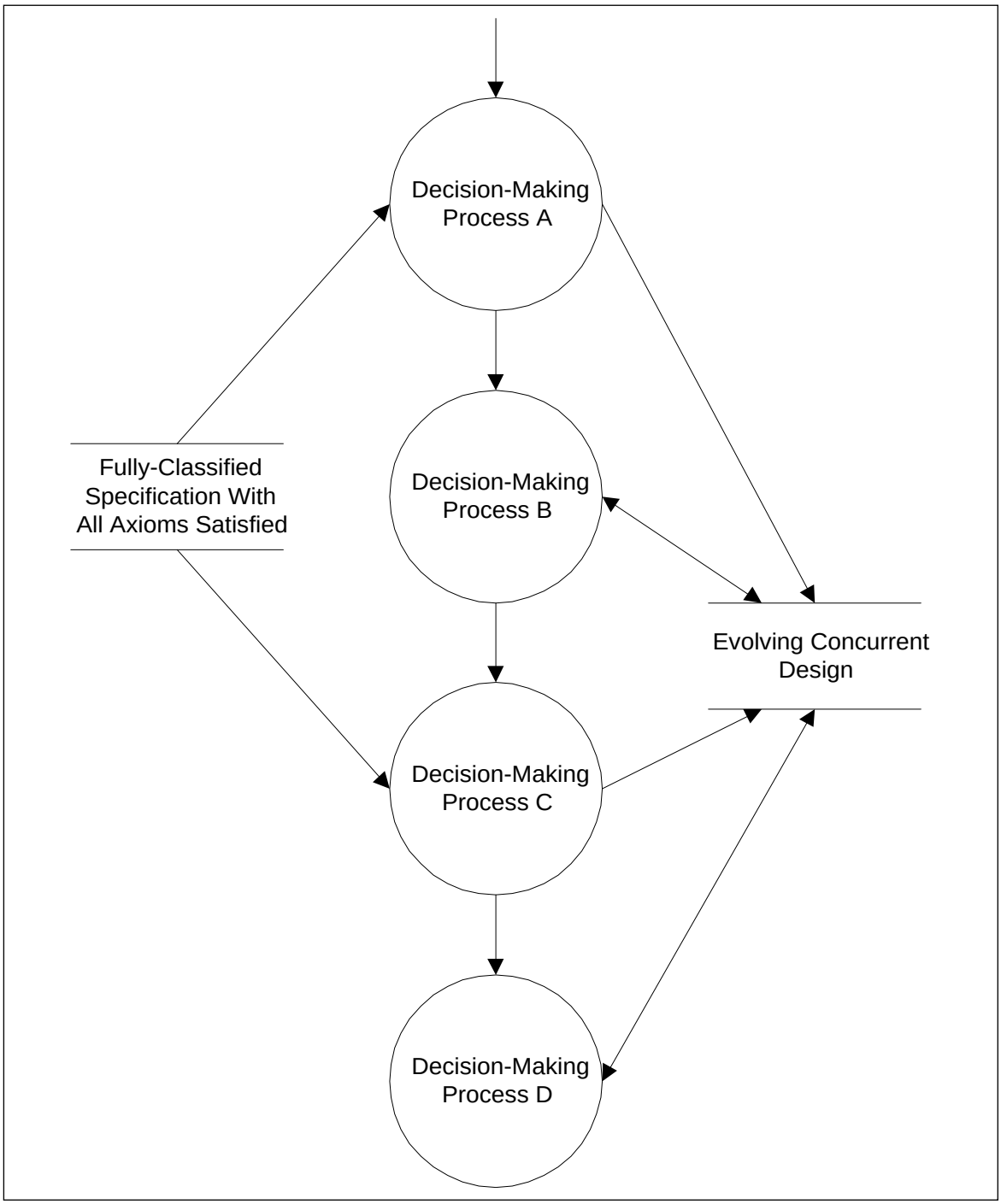

Figure 3. General Organization of Knowledge within a Decision-Making Knowledge Base

approach to organizing the knowledge within a particular, but unspecified, decision-making knowledge base.

The decision-making knowledge base illustrated in Figure 3 is organized as four, distinct, decision-making processes, invoked sequentially from the top (Decision-Making Process A) to the bottom (Decision-Making Process D). As shown in Figure 3, decision-making processes can use information from a valid specification, as described in Chapter 4, (that is, a specification where all concepts are classified fully and all applicable axioms are satisfied) and from the evolving, concurrent design. Notice also that decision-making processes can update the evolving, concurrent design, but not the specification. All elements of the evolving, concurrent design are represented as instantiations of entities and relationships defined within the Design Meta-Model, as described in Chapter 5.

Within each decision-making process, design-decision knowledge is expressed through a set of rules, each representing a design decision that might be taken. The general form for any design-decision rule is as follows.

Rule: Name Of The Rule (optional preferred ordering)

if

a situation is recognized in the specification and/or the evolving, concurrent design

then

1. take an appropriate action to alter the design
2. capture the rule that made the decision, the action taken, and the rationale
3. where appropriate, establish or update any traces between specification elements and design elements

fi

Each rule has a unique name, and is expressed in the form: **if** antecedent **then** consequent **fi**. The antecedent of a rule recognizes the existence of some situation in the specification, in the evolving, concurrent design, or in both. The antecedent of a rule is satisfied whenever the situation recognized by the rule exists within a specification and/or an evolving, concurrent design. The antecedent of a rule is unsatisfied whenever the situation recognized by the rule does not exist within a specification and/or an evolving design. The design-decision rules composing any decision-making process can cycle repeatedly between being satisfied and unsatisfied while the process is active.

Each decision-making process, then, is modeled as a cyclic set of decision points. At every decision point, each design-decision rule with a satisfied antecedent represents a decision eligible to be taken. At each decision point, one eligible decision is selected and acted upon. Then the antecedents for all design-decision rules are reevaluated and the decision-making process cycles to the next decision point. This decision-making process continues until reaching a decision point where the antecedents of no design-decision rules remain satisfied, and then the decision-making process terminates.

Generally, at each decision point, no design-decision rule takes precedence over any other rule; thus, when the antecedents of multiple rules are satisfied at a decision point any one of those rules can be selected and acted upon. In some situations, however, certain design decisions might be preferred over others. For example, a designer might prefer to combine a periodic input task with a control task to which the input task provides the sole input, rather than to combine that input task with other periodic input

tasks having the same period. In the absence of these conflicting possibilities the designer might prefer simply to combine the input task with other periodic input tasks having the same period. In another case, a designer might reserve a default decision to be made only in the absence of other decisions being available. Situations such as these can be accommodated by annotating the appropriate design-decision rules with an optional preferred ordering, as shown in parentheses in the preceding example rule. The presence of preferred ordering within a set of design-decision rules modifies the selection process at decision points within the decision-making process. At each decision point, the rule with the greatest preference is selected from among the rules with satisfied antecedents. The selected rule is then acted upon. This cyclic model that controls each decision-making process is illustrated through an algorithm shown as Figure 4.

3.5 Proof-of-Concept Prototype

Evaluation of the proposed approach for assisting a human designer to generate concurrent designs from data/control flow diagrams is addressed through a proof-of-concept prototype that is applied to four design problems. The proof-of-concept prototype is described in Chapter 10. Application of the prototype to an automobile cruise control and monitoring system is illustrated in Appendix B. Appendix C presents several solutions, as generated by the prototype, for a robot controller problem. Appendix D describes a number of concurrent designs for an elevator control system, as generated by the prototype. Appendix E shows how the prototype helps to generate a

```
begin-procedure Decision-Making Process

 begin-definitions
  design-decision-rules  /* The set of all rules specified within this
                            decision-making process.
                         */
  active-rules  /* The subset of design-decision-rules with
                   satisfied antecedents.  This subset is maintained in a
                   partial order from most preferred ordering to least preferred
                   ordering as specified in the design-decision-rules.
              */
 end-definitions


 begin-procedure evaluate rule antecedents
   remove all members from the set of active-rules
   for every member of design-decision-rules
      if the antecedent of the member is satisfied
      then add the member to active-rules
     endif
   endfor
 end-procedure evaluate rule antecedents



 begin    /* Cyclic Decision Points */
     perform-procedure evaluate rule antecedents
     while the set of active-rules is not empty
       select the first rule from active-rules
       execute the rule
       perform-procedure evaluate rule antecedents
     endwhile
 end      /* Cyclic Decision Points */

end-procedure Decision-Making Process
```

Figure 4. An Algorithm Representing the Cyclic Execution Model Controlling Each Decision-Making Process

concurrent design for a remote temperature sensor. An evaluation of the proposed approach is provided in Chapter 11.